Separation Platform for Integrating Complex Avionics (SPICA) Final Report

December 15, 2013

SPONSORED BY: NASA Ames Research Center

Contract #: NNX13AB99A

Contract Period of Performance: 16 November, 2012 to 15 November, 2013.

Contractor: Adventium Enterprises, LLC
Principal Investigator: Dr. Mark Boddy
Business Address: 111 3rd Ave S, Suite 100
Minneapolis, MN, 55401
Phone Number: 651.442.4109

Contents

1 Introduction

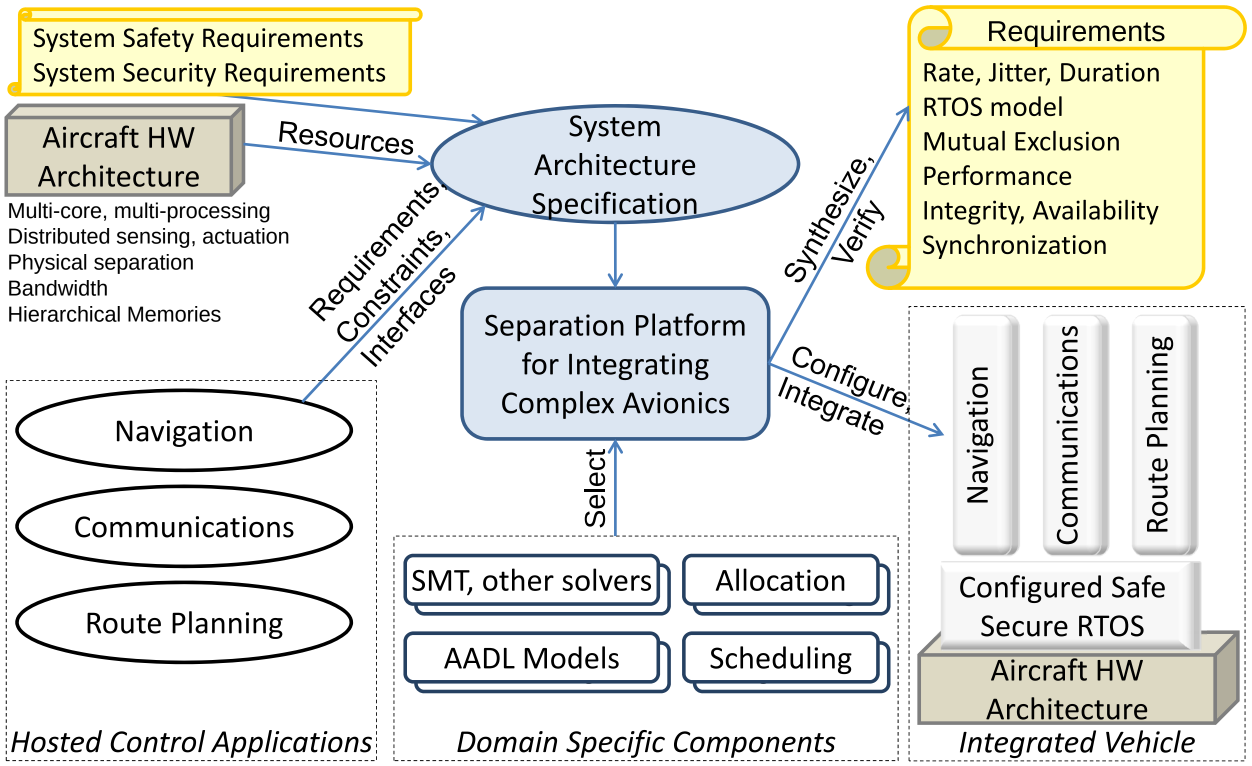

Figure 1: SPICA plays a key role in avionics system design and integration.

Modern avionics systems must support a large and rapidly increasing number of diverse, mixed-criticality functions, deployed on increasingly complex and diverse avionics hardware, while meeting stringent performance requirements. Effective use of multi-core and distributed avionics systems is hindered by the difficulty of allocating processor, memory, and communcation resources among the required functions. Building on previous work in constraint-based resource allocation, scheduling, and configuration management for fault-tolerant, real-time systems, Adventium's Separation Platform for Integrating Complex Avionics (SPICA) system addresses this need by generating static schedules and time-phased resource allocations for distributed, hierarchical, and mixed-criticality systems. Treating the whole aircraft as a *system*, rather than as a collection of federated components, SPICA enables effective use of multi-core processors and advanced communications networks, resulting in improvements in the performance, efficiency, safety, and dependability of complex avionics systems.[1]

In this report, we describe the goals, technical approach, and results of Adventium Labs' SPICA Phase I project, funded under the National Aeronautics and Space Agency (NASA)'s Leading Edge Aeronautics Research for NASA (LEARN) program. The key research objective of the Separation Platform for Integrating Complex Avionics (SPICA) is to system-

[1]As defined in [1], dependability includes reliability, availability, safety, maintainability, confidentiality and integrity.

atically address *aircraft-level* avionics design and integration challenges, from early in the requirements and design processes through integration and test. SPICA specifically addresses: **real-time requirements,** such as rate, duration, jitter, and end-to-end latency; **hierarchical architectures,** including cabinets, modules, processors, cores, and their interconnections at all levels; **incremental reconfiguration** for upgrades, fault tolerance, fault response and guaranteed margin; **time-phased global resource assignment** of functions at multiple criticalities. SPICA guarantees separation, dependability, availability and integrity, as well as supporting graceful degradation, mission assurance, and survivability.

Our Phase I program addressed the primary risks of this approach, including complexity-driven scaling in domain-specific solver performance and the interaction between problem requirements (e.g., separation constraints) and the allocation flexibility provided by a given avionics architecture. In the Phase I effort we have generated a formal specification of the complete set of constraints, sufficient to represent a wide range of different avionics architectures and problems. We have produced a large set of test problems for input to the `yices` Satisfiability Modulo Theory (SMT) solver, demonstrating the use of those constraints, along with output results and performance data, as well as a tunable test problem generator, automatically generating problem instances in `yices` input format. Finally, we have also produced an exemplar aircraft avionics architecture, rendered in both diagrams and Architecture Analysis and Design Language (AADL).

Current system performance is adequate to testing, but not sufficient for a mature system. The current prototype will solve problems involving dozens to hundreds of constraints, in minutes. Experiments have demonstrated that the growth in solving time for increasing problem size is reasonable: there is no sharp "knee" in solution times. SMT technology has demonstrated performance on millions of constraints, and in previous work, we have demonstrated several orders of magnitude increase in problem size and decrease in solving time, using a combination of problem reformulation and search control.

1.1 Innovation and Novelty

In current aviation platform development and upgrade processes, integration is performed during late-stage implementation phases, in testbeds, system integration laboratories, and even in operational systems. As shown in Figure 2, this is a very costly way to do integration due to the rapidly increasing cost to fix problems as they are detected later in the design process. The key innovation of SPICA is to support early-stage (ideally design-phase) *aircraft-wide* integration by providing and then maintaining guarantees that temporal and other performance requirements will be met by the system as a whole. This reduces late-stage performance surprises and delays, as well as reducing initial and lifecycle costs of aeronautic systems.

Applying constraint-based approaches, including the use of SMT solvers, to scheduling and allocation problems is not a new idea. The innovation in SPICA is to apply this technology to generate a *static allocation*, derived from a complex, hierarchical, mathematically-correct model of avionics systems as represented in AADL. Related work in this area includes the application of SMT solvers to Aeronautic Radio Incorporated (ARINC) 664 communications scheduling (but only to communications) as in [10], as well as a variety of tools for *online, dynamic* scheduling. In contrast, SPICA emphasizes the formulation of a set of con-

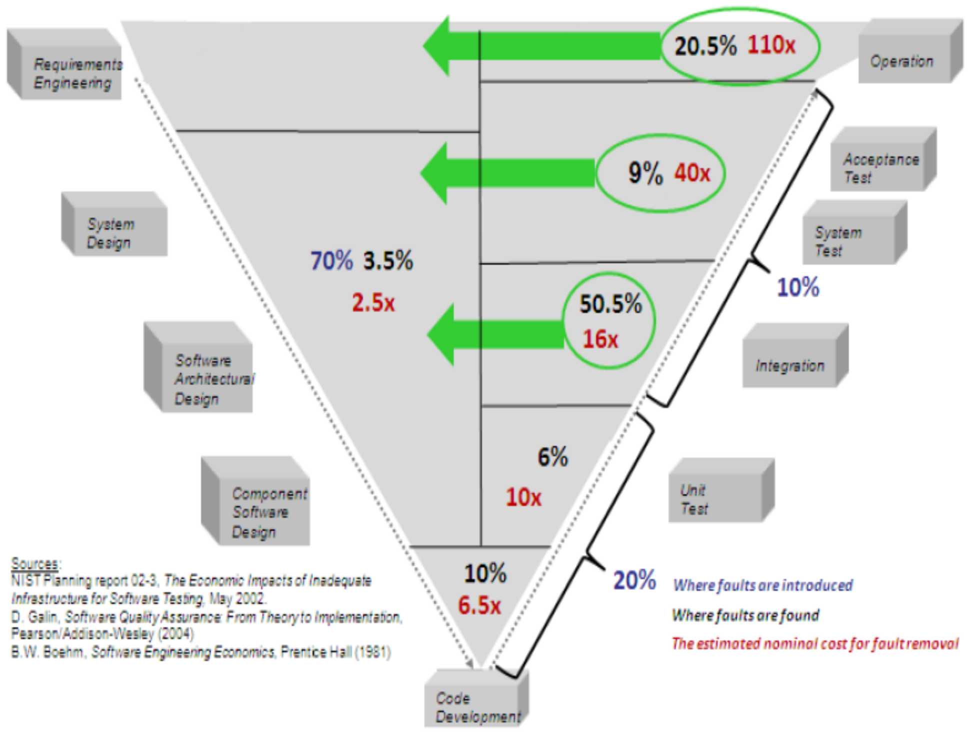
Requirements Engineering
System Design
Software Architectural Design
Component Software Design
Code Development
Unit Test
Integration
System Test
Acceptance Test
Operation
20.5% 110x
9% 40x
70% 3.5%
2.5x
50.5%
16x
6%
10x
10%
6.5x
10%
20%
Where faults are introduced
Where faults are found
The estimated nominal cost for fault removal
Sources:
NIST Planning report 02-3, The Economic Impacts of Inadequate Infrastructure for Software Testing, May 2002.
D. Galin, Software Quality Assurance: From Theory to Implementation, Pearson/Addison-Wesley (2004)
B.W. Boehm, Software Engineering Economics, Prentice Hall (1981)

Figure 2: Most bugs are introduced early in the design process, and discovered late in implementation, integration, and even deployment. The cost to fix these bugs grows dramatically as detection is delayed [5].

straints defining a *correct* time and space allocation of computation, communication, and memory resources, and then solves to find that allocation. The most closely related work is our previously-developed scheduler for the time-and space-partitioned Integrated Modular Avionics (IMA) architecture deployed on the Boeing 777, 717, and Lockheed C5-AMP. That scheduler did not address allocation of functions to processing elements (and thus memory), allocation of functions to partitions, or ARINC 664 communications scheduling (which had not been published at that time). In more recent work, we have also developed a proof-of-concept mixed-criticality, real-time scheduler for a multi-core, hypervisor-based system, called MICART [7].

1.2 National Aeronautics Impact

SPICA's potential impact on NASA and national aeronautics challenges is to provide a novel capability for rapid, dependable integration of new functions, systems, and components into new and legacy avionics architectures. SPICA will provide this while maintaining performance guarantees of existing capabilities and systems, enabling certified, flight critical avionics functions to safely and robustly operate side-by-side on shared hardware with functions not certified at the same level, without having to be re-certified as new functions are added. SPICA will also enable fuller utilization of multi-core processors and embedded systems while maintaining these guarantees, improving the performance, efficiency, safety, and dependability of aeronautical avionics. Specifically with regard to NASA, SPICA is relevant to the following programs:

Integrated Systems Research – SPICA will provide support for investigating the avionics-level integration of novel functions, hardware systems, and architectures.

Aviation Safety – This program covers assurance for flight-critical systems, including managing the complexity of architecting, validating, and verifying the correct functioning of increasingly complex avionics. SPICA's output is a concrete schedule which can easily be verified to satisfy requirements governing execution times, latencies, and sampling rates, as well as more complex issues such as metastable communications across an asynchronous boundary.

Orion – SPICA is developing relevant capabilities for other complex, networked vehicular systems. For example NASA's Orion Multi-Purpose Crew Vehicle (MPCV) uses several of the protocols and standards SPICA is designed to address.

In this report, we summarize the progress on the Phase I SPICA project (Section 2. We then introduce the formal constraint model we have developed, in Section 3. Section 4 presents our notation and definitions, while Section 5 contains the constraints that define the SPICA scheduling problem. Section 6 uses those constraints to define a family of scheduling problems that can be addressed by SPICA, with some simple examples provided in Section 7. Finally, in Section 8 we present our conclusions and describe the next steps, specifically including our objectives for the Phase II project.

2 Phase I Progress and Results

In Phase I, we analyzed the features of modern avionics architectures, with primary emphasis on the Boeing 787 Common Core System (CCS) and the Airbus 380 Integrated Modular Avionics (IMA) architectures. Based on that, we synthesized a reference problem and a suite of test examples, with an emphasis on IMA, distributed sensing and actuation, and modern avionics communications networks (e.g., time triggered, globally asynchronous locally synchronous). We generated a formal specification of a set of constraints sufficient to represent both the 787 CCS and the 380 IMA, as well as a wide range of other avionics architectures. In particular, resource allocation for ARINC standards 653, 659, and 664 is fully supported in SPICA, including the representation of hierarchical processing modules, asynchronous boundaries, and multi-hop networks.

Figure 3 shows the avionics functions that we have confirmed can be modeled in AADL (left column), the corresponding software features that have been specified in the formal constraint model (center column), and hardware features that can be represented in AADL and have been specified in the formal constraint models. All of the software and hardware features listed in Figure 3 have been implemented in the Phase I SPICA prototype.

As a specific instance, Figure 4 shows the constraint model implemented in SPICA to support partitioning and preemption in ARINC 653 systems. Among the features that can be seen in that figure are the differences in context-switch time between inter-partition and intra-partition context switches, preemption of one partition by another, and explicit representation of the partition cleanup time once an interrupt is received.

The architecture of the Phase I SPICA prototype implementation is shown in Figure 5.

Applications
FMS
Displays
Graphics
Data Communication Gateway
Data Recorders
Health Management
Primary Flight Controls
Actuator Control Electronics
Radios
Inertial Reference Units
Surveillance
Fuel Management
Landing Gear

Software
Partition
Context Switch
Exclusion
Preemption
Partition Assignment of Tasks
Tasks
Jitter
Context Switch
Required Computational Duration
Initial load time
Minimum Duration
Period
Preemption
Allowed Binding
Exclusion
Flows
Latency
Jitter
Size
Oversampling
Undersampling
Redundancy

Hardware
Processors
Memories
Communication Buses
Asynchronous Boundaries
Modules
Cabinets
Aircraft

Figure 3: All of the applications listed can be represented in AADL. All of the software features needed to model those applications have been specified in the formal constraint model. All of the hardware features shown can be represented in AADL and have been specified in the formal constraint model.

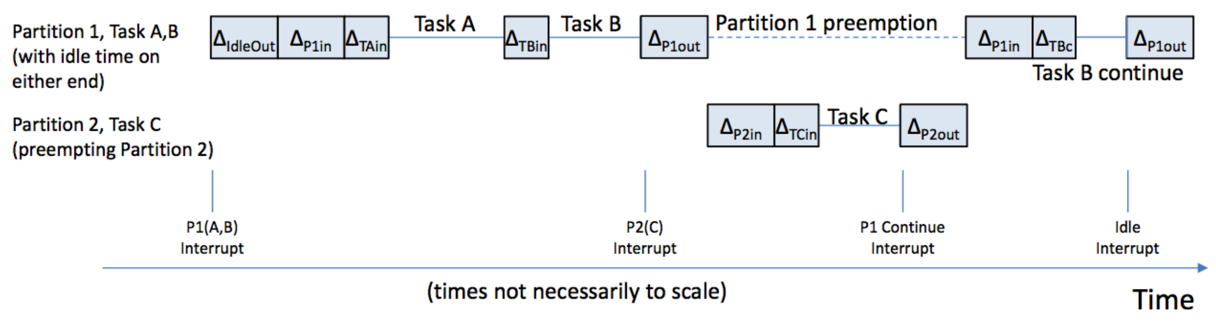

Figure 4: SPICA constraint model for ARINC 653, showing two partitions (1 and 2), and three tasks (A, B, and C). Partition 2 preempts Partition 1. All of the required context switch times are modeled.

With the exception of the two lighter-colored arrows on the lower left, all of the functions, representations, and interfaces shown have been implemented. Consequently, we place the prototype at a Technology Readiness Level (TRL) of 3. By the completion of the Phase

II effort we expect to attain a TRL of 5 or 6, depending in part on whether or not our proposed optional task is funded. The Open Source AADL Tool Environment (OSATE) can be used to generate or modify an AADL model, containing any or all of the elements previously discussed. The SPICA prototype includes an OSATE plugin that automatically extracts a constraint model from an AADL model, representing the output in the constraint language Minizinc.[2] As shown by the lighter-colored arrows, we then have the option of either implementing a translator from Minizinc into the python representation from which we generate input for the `Yices` SMT solver[3], or of modifying the OSATE plugin to generate that python representation directly.[4] Our current plan is to modify the plugin.

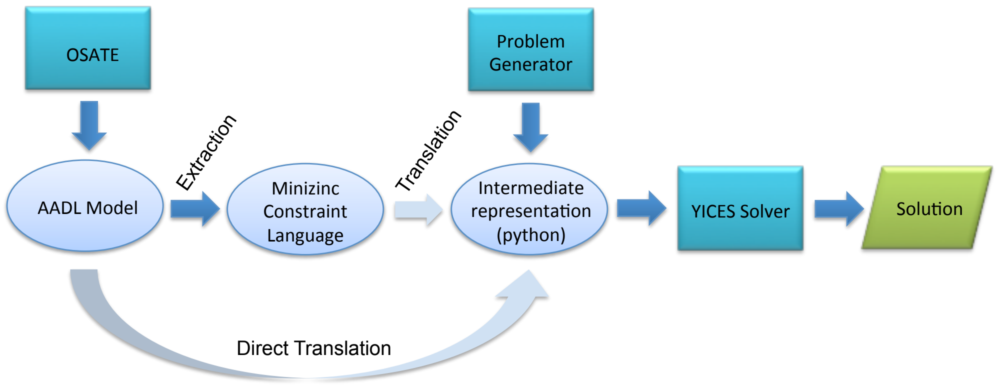

Figure 5: SPICA Phase I implementation

Generating python code directly from the AADL model has two advantages. First, we avoid any "semantic gaps" caused by translation through Minizinc. And second, we have more direct control over the python generated, and thus over the input to `Yices`. Problem instances for input to the `Yices` solver may also be generated by a problem generator we have implemented for testing purposes as part of this project.

Finally, the `Yices` solver is invoked, generating either a solution (i.e., a time and space allocation of computing, memory, and communication resources which satisfies all of the input constraints), or an indication that the constraints are mutually inconsistent, signalling that no allocation can be found for the current design. Figure 6 presents an allocation found by SPICA for a moderately-complex input problem, consisting of a few dozen tasks and constraints. Among other interesting features, this particular example demonstrates enforcement of latency constraints across an asnynchronous boundary.

In order to evaluate both the breadth and correctness of our constraint model and the scaling performance of SPICA, we generated a large set of test problems (effectively a set of *unit tests* for SPICA), as well as implemented a tunable test problem generator, which automatically generates problems of various sizes, with specified types of constraints present. The current prototype will solve problems involving dozens to hundreds of constraints in minutes. This performance is adequate for testing, but not for a mature system.

[2]This plugin was implemented on a previous project.

[3]http://http://yices.csl.sri.com/

[4]We generate python that expands into `Yices` input format rather generating that input format directly because the python can be much more concise and structured, and so easier to interpret, debug, and modify if needed.

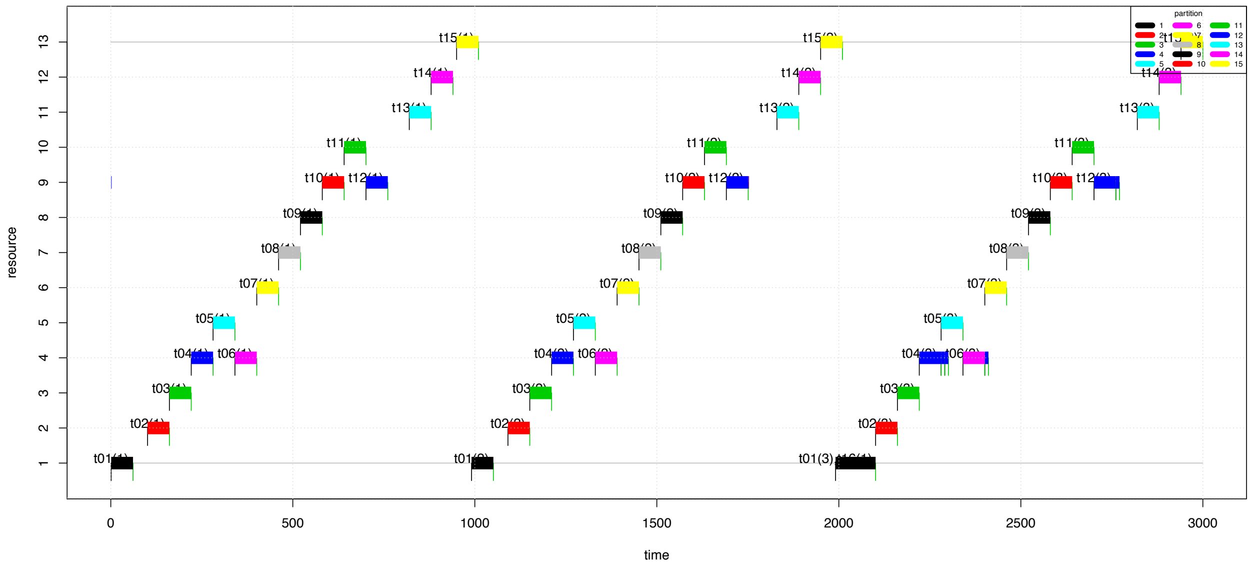

Figure 6: Graphical display of a time and space allocation generated by SPICA

As shown in Figure 7, the growth in solving time for increasing problem size is reasonable: there is no sharp "knee" in solution times. We are confident in being able to scale SPICA up to the required problem size. SMT technology has been demonstrated on millions of constraints [9], and in previous work, we have shown that a combination of problem reformulation and search control can decrease solving time and increase the size of problems handled by several orders of magnitude.

3 Preliminaries

In the next few sections, we present the constraint formulation we have developed for SPICA. This formulation supports temporal scheduling, resource allocation, and planning. These constraints take a different form than the deadline/offset problems of [2] and [8]. Using mathematical notation for the constraints simplifies the process of translating our models into an SMT language which can then be fed to an SMT solver in order to generate a solution. Previous work in this area has used different flavors of constraints or otherwise failed to capture the full richness of the constraints required for ARINC 653. Our work here was inspired by the SMT modeled job-shop scheduling of [4] while the ARINC constraints we place upon those were discussed in [3].

The contributions of this work are twofold.

1. Present a unified source for all of the ARINC type constraints, which must be taken into account by a system designer, written in a precise, mathematical form.
2. To present the constraints in such a form that they can be presented to an SMT solver and thereby, present an efficient ways to produce a static schedule which satisfies all of the ARINC type constraints.

In Section 4 we present the definitions we have found useful in formulating a scheduling problem, including examples and discussion. Section 4 is further subdivided into hardware

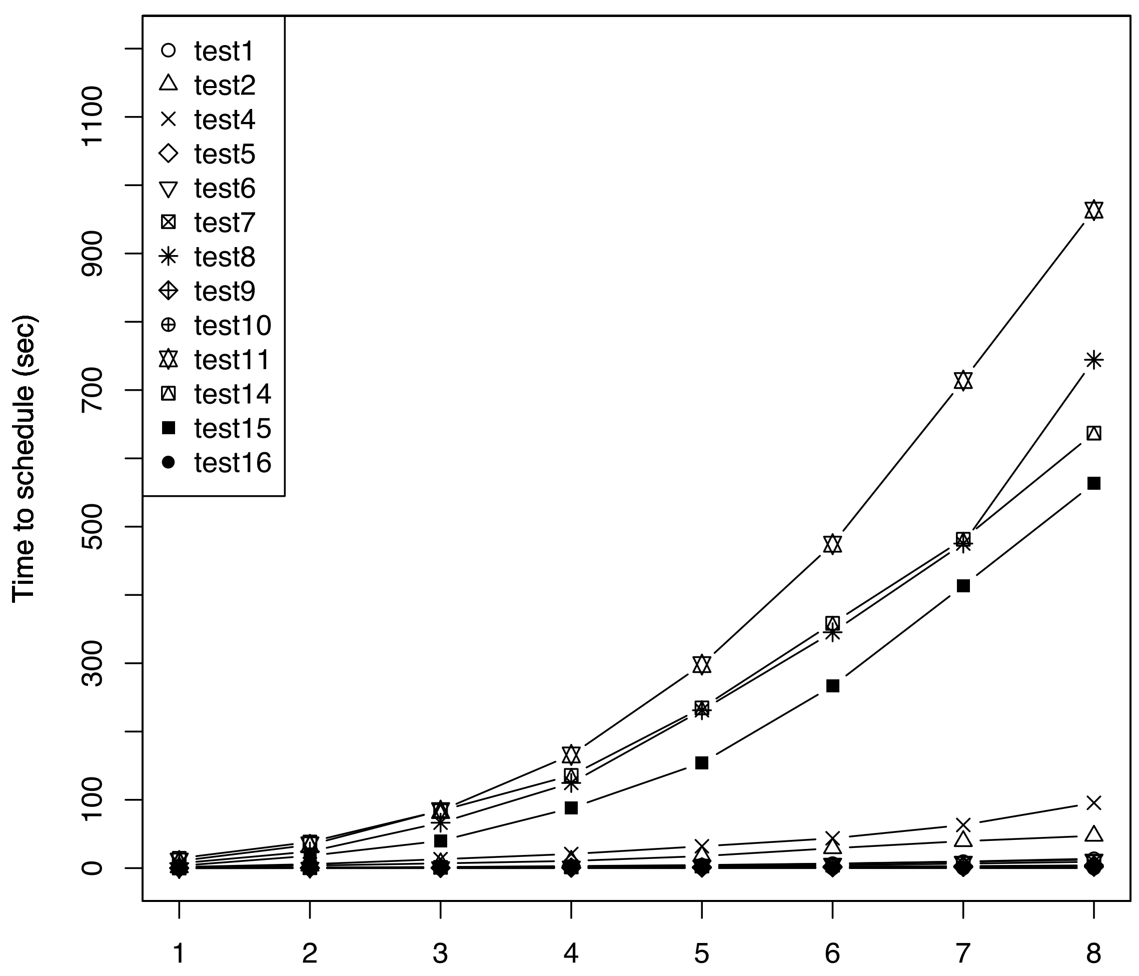

Figure 7: Solution times increase in a well-behaved way with increasing problem size. The horizontal axis is a multiplier for each problem. So, the instance of "test8" at horizontal coordinate 5 is 5 times as large as the smallest instance.

(resources and systems) and software (tasks and partitions) sections. Resources represent the assets required to perform that computation, bound together and networked with systems (inspired by AADL), while tasks represent discrete aspects of computation. Section 4 concludes by abstractly defining the ways in which tasks may be mapped onto resources both temporally and spatially. We refer to these mappings as as *schedules* and *allocations* respectively, and taken together they will be the primary focus of this document.

Throughout this document time and all measurable quantities are assumed to be non-negative integers ($\mathbb{Z}_{\geq 0}$) and all sets are assumed to be finite and non-empty, unless otherwise stated. Taking a system designers perspective, we assume that the quantization of resources is provided to us in advance, and since we are using mathematical notation and smt solvers, the quanta does not significantly effect the tractability of the solution.

4 Notation and Definitions

This section sets out some basic definitions for concepts such as resources, tasks, flows, and schedules.

4.1 Resources

The starting point of our discussion is to describe the resources which allow computation to be performed and work to be done. For our purposes resources will represent the assets of a computing system which must be explicitly allocated and scheduled and will all be metric in nature. Examples include processors, memory, and communications channels. Resources may be partitioned both in time and capacity, as they will used by multiple tasks.

Definition 4.1 (Resource) *A* Resource *is a distinguished facility for performing computation which has some available capacity ξ (with units unspecified).*

Distinct resources are given distinct names, even when the resource provide facilities to perform the same type of work, for instance two Central Processing Units (CPUs) which one desires to schedule independently could be modeled as two different resources, while two cores of a single CPU might be modeled as a single resource with capacity 2.

Resources come in many different shapes and sizes but for our purposes in this document the resources will be one of two flavors, *fluent* (whose full capacity can be used by different tasks at different times, making them subject to Constraint 5.14) and *static* (which must be allocated for all time, and thus are subject to both of Constraints 5.14 and 5.15). We assume here that all resources are fluent and pick out a distinguished set of static resources, which we typically refer to as R_{static}

For clarity we will sometimes use functional notation to refer to the capacity of a resource R.

$$\xi(R) = \xi_R$$

In the traditional scheduling literature [8] [2], resources are unitary and can be modeled as a resource with capacity $\xi(R) = 1$. A typical collection of resources for our purposes would be processors, communication busses, and peripheral devices.

When we discuss scheduling problems, it is generally assumed that resources are taken from a pre-determined set of available resources. We will typically refer to this pool as $\mathcal{R}$ which we call the *set of available resources* with a distinguished subset of static resources $R_{static} \subseteq \mathcal{R}$

4.2 Systems

Resources themselves do not exist in isolation but are typically grouped together into aggregations such as a desktop computer or rack mounted server, consisting of one or more processors, cache, ram, network, and other peripheral devices. To reflect this we introduce the concept of a *system*. Our definition is inspired by IEEE-1471 [6], which characterizes systems as “A collection of components organized to accomplish a specific function or set of functions”.

Definition 4.2 (System) *A* system *S is a non-empty set of interrelated resources (themselves non-empty sets) and other systems.*

The idea of a system is meant to capture the concept of a bundle or grouping of resources that must be used together to perform some useful work. For example, a computer system (host) might be an aggregate of one or several processors, memory, and an Ethernet interface, and could consist of several subsystems such as a non-networked subsystem consisting of only a processor and memory, as well as a networked subsystem consisting of processor, memory, and ethernet, allowing the designer to model a variety of ways for the software to interact with the hardware. Under this definition a non-networked computation could run on the subsystem of processors and memory. This illustrates an important point, which is that systems can be hierarchical, allowing for complicated groupings of resources as well as encoding a network topology. Example 4.3 illustrates both grouping and network connections.

Resources and systems are fixed by the system designer's choice of hardware. For example a system architecture might offer K processors, M megabytes of memory, N high-speed networks and so forth. Some resources, e.g. networks, can be shared among multiple systems.

All of the resources within a given system are assumed to be synchronous with globally time triggered events and with no timing drift. It is important to note, however, that this assumption does not extend across systems. Distinct systems are allowed to be asynchronous with clock skew. We deal with the asynchronous nature of systems by virute of flows (See Constraint 5.10). Flows are the only constraint which must be applied across systems, so we predict the worst case latency for a flow which crosses an asynchronous boundery and adjust our latency constraint so that the desired observed latency is always obtained (or a system is considered unschedulable).

The mechanism we provide to detect asynchronous boundaries is a function $sync : \mathcal{R} \times \mathcal{R} \to \mathbb{Z}_{\geq 0}$ which records the number of asynchronous boundaries between two systems.

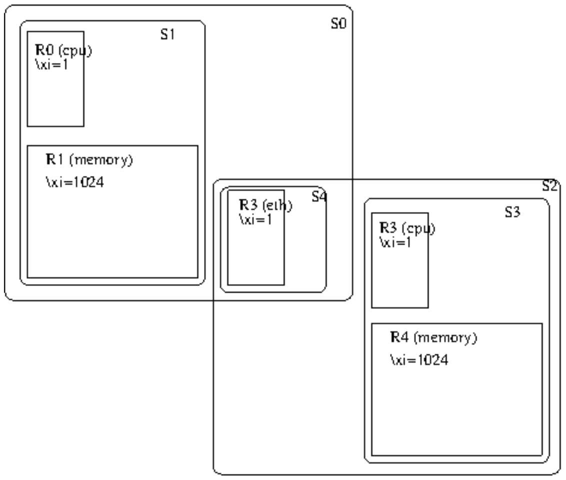

Figure 8: Example of system hierarchy and network topology

Example 4.3 *In this example we are modeling two computers with separate memory and processor resources connected via a shared Ethernet connection. Formally, this example* *consists of 3 fluent resources* (R_0, R_2, R_3) *and 2 static resources* (R_1, R_4)*, which have been divided into two systems. Each system contains a "cpu", a "mem", and a "eth". The systems are connected by the "eth" resource, which is contained in both systems.*

$$\mathcal{R} = \{R_0, R_1, R_2, R_3, R_4\}$$

$$\mathcal{S} = \begin{Bmatrix} S_0 = \{R_0, R_1, R_2\} \\ S_1 = \{R_0, R_1\} \\ S_2 = \{R_2, R_3, R_4\} \\ S_3 = \{R_3, R_4\} \\ S_4 = \{R_2\} \end{Bmatrix}$$

To define the quantity function on these resources, we simply define the quantity function

$$\begin{aligned} \xi(R_0) &= 2, \\ \xi(R_1) &= 1024, \\ \xi(R_2) &= 1, \\ \xi(R_3) &= 2, \\ \xi(R_4) &= 1024. \end{aligned}$$

This is illustrated in Figure 8. Notice in particular that the resources hierarchy is encoded by the subset inclusions $S_1 \subset S_0$ *and* $S_3 \subset S_2$ *while network topology is encoded by the sharing of* R_2 *by systems* S_0 *and* S_2*.*

4.3 Tasks

In this document, *tasks* are an atomic unit of computation and information processing, which could represent actions as diverse as bus transfers, sensor readings, or traditional computation.

Definition 4.4 (Task) *A* Task $\tau = (C, Q, \mathcal{A})$ *is a discrete unit of information processing in which*

- $C : \mathcal{S} \to \mathbb{Z}_{>0}$ *is a (partial) function which denotes the amount of time to budget to run a task on a system (worst case execution time, not including context switch),*
- $Q : \mathcal{R} \to \mathbb{Z}_{>0}$ *is a (partial) function which denotes the quantity of each resource required,*
- *and* $\mathcal{A} \subseteq \mathcal{S}$ *is the set of* allowed systems*, i.e. the systems on which* τ *is allowed to run.*

Compute time and quantity are independent measurements, with quantity being measured in the same units as the resource R. The intuitive idea behind this is that the quantity of a resource required by a task depends on the resource (for instance, a task which requires 4MB of ram), however the compute time depends on all the resources which are made available to a task, for instance, a task might be able to run on two systems containing memory with different latencies or throughput, which would change the compute time of the task.

The functions C and Q for compute time and capacity are made with partial functions to avoid defining a value of C for every possible resource. This can be easily extended to a function on the entire set of resources with the assumption that $C(R) = \infty$ if $C(R)$ is not explicitly defined.

Tasks can be defined in various ways: periodic, sporadic, aperiodic. This work is primarily concerned with periodic tasks.

Definition 4.5 (Periodic Task) *A* periodic task *is a task that must be performed at regular intervals (called the period T) in perpetuity. A periodic task is parameterized by the 4-tuple $\tau = (T, Q, C, J, \mathcal{A})$ in which, $T > 0$ denotes the period, and $J > 0$ denotes the maximum-allowed-jitter. Both period and jitter are intrinsic properties and do not depend on resources.*

For clarity we will sometimes use functional notation for tasks

$$\begin{aligned} C(\tau) &\equiv C \\ T(\tau) &\equiv T \\ J(\tau) &\equiv J \\ Q(\tau) &\equiv Q \\ \mathcal{A}(\tau) &\equiv \mathcal{A} \end{aligned}$$

The notions of period and jitter here are, for now, arbitrary constants. For details on jitter, see Figure 12 as well as Definition 4.7.

The parameters T and J of periodic tasks do not entirely make sense on their own, because we do not schedule tasks, we schedule task instances. Both period and jitter place requirements on the difference in start times between task instances.

Definition 4.6 (Task Instance) *Each occasion a periodic task $\tau = (T, C, Q, J, \mathcal{A})$ executes to completion is known as a* task instance. *Task instances are numbered sequentially, so that the j^{th} release of τ is known as $\tau(j) = (S_j, F_j, P_j)$, with start time S_j, finish time F_j, and preemption count P_j. All task instances of periodic task τ have the same compute time $C(\tau)$ and allowed systems $R(\tau)$, as τ.*

These functional notations can be extended to instances $\tau(j)$ of a task τ

$$\begin{aligned} C(\tau(j)) &\equiv C(\tau) \\ T(\tau(j)) &\equiv T(\tau) \\ Q(\tau(j)) &\equiv Q(\tau) \\ \mathcal{A}(\tau(j)) &\equiv \mathcal{A}(\tau) \\ S(\tau(j)) &\equiv S_j \\ F(\tau(j)) &\equiv F_j \\ P(\tau(j)) &\equiv P_j \end{aligned}$$

We will typically refer to the set of all tasks as $\mathcal{T}$ and the set of all task instances as $\mathcal{T}_{inst}$ (potentially infinite). Note that $F(\tau(j)) - S(\tau(j))$ need not equal $C(\tau)(R)$ as task instances may be preempted. If a task is preempted, it should be reflected in the preemption count $P(\tau(j))$

With start/finish times for task instances, we can now give meaning to the jitter parameter of a task. All references to *jitter* in this document will mean relative start jitter.

Definition 4.7 (Relative Start Jitter) *The* relative start jitter *of a task is the maximum allowed deviation of the start time of two consecutive instances [2, p.73].*

$$J(\tau) = |S(\tau(j+1)) - S(\tau(j)) - T(\tau)|$$

The actual jitter between two task instances is a function of both the schedule (see section 4.6) and physical effects of a given system such as clock skew and asynchronous boundaries. Constraints on allowable jitter help define a *feasible* schedule (see Section 6). In some cases jitter will be unimportant, and the constraint on jitter may simply be specified as ∞.

4.4 Partitions

Partitions support grouping of related tasks to run in a lightweight but isolated and protected common runtime, saving context switch time, but potentially increasing latency and jitter, as these can only be computed with respect to partition execution, rather than execution of individual tasks within a given partition.
Definition 4.8 (Partition) *A partition $P = (T, \Delta_{in}, \Delta, \mathcal{A})$ is a set T of tasks which are required to run on the same set of resources (system). The list of systems on which they are allowed to run mirrors the allowed systems of tasks, and is denoted $\mathcal{A} \subseteq \mathcal{S}$.*
It is considered invalid to have a task in two partitions. Given a partition P we introduce functions

$$\begin{aligned} T(P) &= T \\ \Delta_{in}(P) &= \Delta \\ \Delta(P) &= \Delta \\ \mathcal{A}(P) &= \mathcal{A} \end{aligned}$$

Because the tasks assigned to partitions are periodic, a single partition may execute within multiple, disjoint intervals across the schedule. Individual slices of execution for a given partition are referred to as partition *activations.* Individual partition activations may also be *preemptible.* Consequently, partitions have two context switch times associated with them, an initial context switch time Δ_{in}, which must be reserved at the beginning of each partition activation and a continuation context switch time Δ which must be allocated whenever a partition activation returns from a preemption (see Figure 9).
Definition 4.9 (Partition Activation) *A activation $\wp$ of a partition P is a* 3*-tuple* $(B, E, \tilde{T})$ *in which $\tilde{T} = \{\tau_i(j)\}$ denotes a set of task instances taken from $T(P)$ (that is, $\tau_i \in T(P)$). The parameters $B, E \in \mathbb{Z}_{\geq 0}$ are the* Begin *and* End *time of the activation.*
In ARINC 653, what is scheduled are partition activations, with task instances *released* at the start of a given partition activation. Consequently, any timing constraints related to tasks or task instances must instead be applied to the start and end times of the appropriate partition activations.

As with tasks, partition activations also inherit the properties of the partition. We use a minor abuse of notation in order to refer to the partition activation $\wp$ associated with a

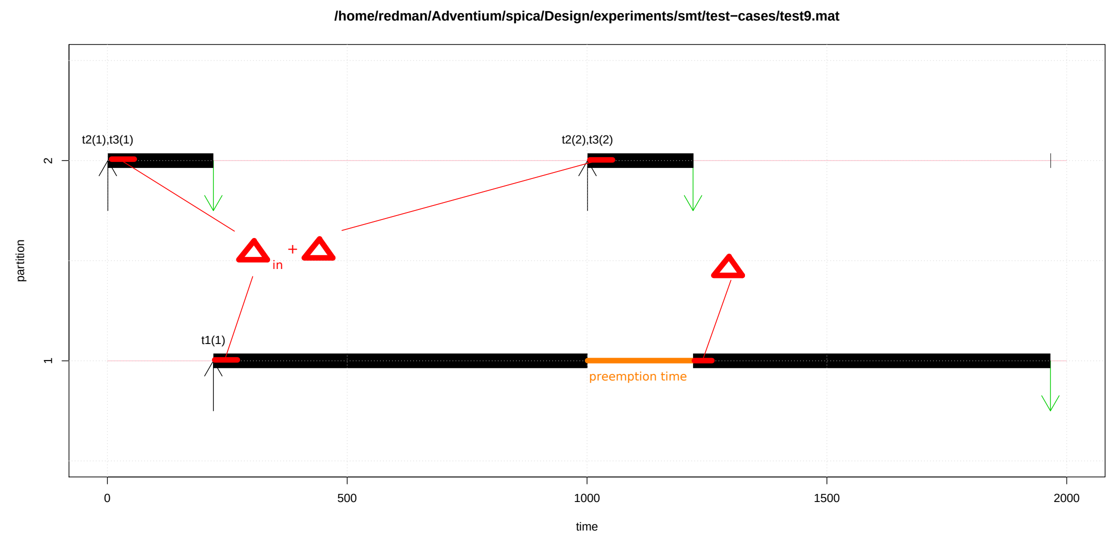

Figure 9: initial context switch, context switch, and preemption time

task instance $\tau(j)$ (i.e., the partition activation at the start of which $\tau(j)$ is released.

$$\begin{aligned} B(\wp) &= B \\ E(\wp) &= E \\ \tilde{T}(\wp) &= \tilde{T} \\ \Delta(\wp) &= \Delta \\ \wp(\tau_i(j)) &= \{\wp \text{ such that } \tau_i(j) \in P\} \end{aligned}$$

We denote the set of all partition activations as $\mathcal{P}_{act}$. If the partition activation for a given task instance is preempted, that task instance may continue execution in subsequent activations for the same partition.

Partitions are meant to provide spatial and temporal resource protection, which we primarily use to provide criticality guarantees, but also can be used to decrease context switch time by providing a lightweight threading mechanism. Tasks which run within a partition must only pay a lightweight context-switch time for the task, while switching between tasks in separate partitions requires the additional Δ of partition level context switch time, the magnitude of which depends on the system on which P is running.

4.5 Flows

Flows model how information moves through a system, imposing orderings and time constraints on tasks.

Definition 4.10 (Flow) *A* flow *is a tuple $\varphi = (g, L)$ where $g = (V, E)$ is a directed, acyclic graph (DAG) whose nodes are tasks and edges form a partial order on tasks. $L > 0$ denotes* end-to-end *latency (see Figure 10 for details).*

As with tasks and partitions, the equivalent functional notation is:

$$\begin{aligned} L(\varphi) &\equiv L \\ g(\varphi) &\equiv g \\ V(\varphi) &\equiv V \\ E(\varphi) &\equiv E \end{aligned}$$

Finally, we will use the term *length* (denoted $\ell(\varphi)$) to refer to maximum graph-theoretic distance in $g(\varphi)$.

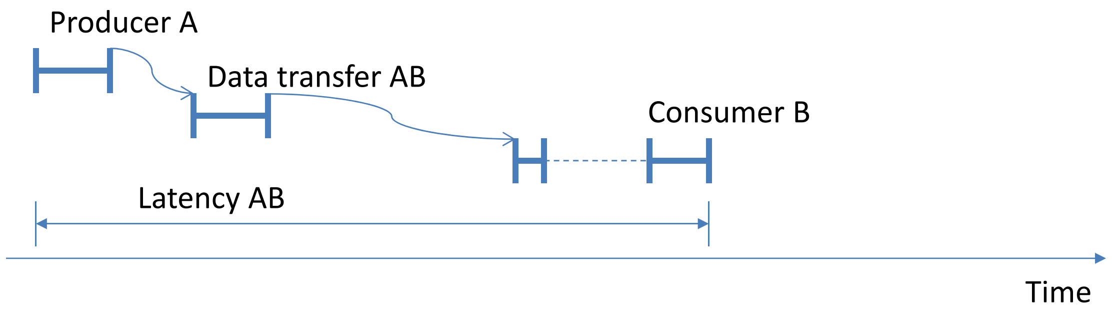

Figure 10: Measuring AB latency is defined from the start of instance $\tau_A(i)$ to the finish of $\tau_B(j)$.

A *Flow* comprises a set of precedence constraints among tasks, some of which represent actual computation, while others represent communication as reserved time on a bus. Typically, we are most interested in end-to-end latency across (part of) a flow, as shown in Figure 10. As task instances provide a specific implementation of tasks, *flow instances* do the same for flows.

Definition 4.11 *A* Flow Instance $\varphi(j)$ *is a flow* φ *in which each node in* g *has been replaced with a task instance.*

Deciding which task instances to include for each task in a flow is part of the problem of generating a feasible schedule.

Example 4.12 *A simple flow with latency 100 between tasks* τ_1 *and* τ_2*, each with* 0 *jitter and period* 100 *would be formalized as:*

$$\varphi = (\tau_1 \rightarrow \tau_2, 1000)$$

An instance of this flow could take the form

$$\varphi(1) = (\tau_1(1) \rightarrow \tau_2(35), 1000)$$

However, this flow instance will not be schedulable, as $\tau_2(35)$ *will start* 3500 *units after* $\tau_1(1)$*, leaving no hope of finishing with the desired latency.*[5]

This example illustrates the need to be somewhat more precise in our discussion of flow instances. In particular, how flow instances map over task instances is determined by the

[5]Recall that this notation refers to the first instance of task τ_1 and the thirty-fifth instance of task τ_2.

relative periods of subsequent tasks in the flow, by flow latency constraints, and by assumptions about how data transfers along the flow are *buffered*.

A common assumption, and the one we employ here, is that there is a single buffer for each node in the flow graph with one or more outgoing edges. In other words, the information generated by a given task instance is preserved, but only until the next instance of that task begins execution. Other assumptions about information transfer lead to different sets of constraints.

Suppose we have a simple flow, involving two tasks with the same period. In this case for any edge $e \in g$ of the form $\tau_1 \rightarrow \tau_2$, we require a flow instance to have the form

$$\tau_1(i) \rightarrow \tau_2(i)$$

or

$$\tau_1(i) \rightarrow \tau_2(i+1)$$

For *cyclic* schedules, as are very frequently found in avionics systems, the arithmetic is modulo the number of instances of each task in the frame. In other words, the "next" task instance for the last producer task may be the first instance of the consumer task in the next frame execution. This constraint enforces regular interleaving of flow instances over task instances, which satisfies the single buffer assumption.

For flows involving tasks with different periods, we must address *oversampling* and *undersampling*. Oversampling occurs when a flow contains the edge $\tau_1 \rightarrow \tau_2$ and $T(\tau_2) \leq T(\tau_1)$. In this case we require that the oversampling rate is an integer, that is

$$\frac{T(\tau_1)}{T(\tau_2)} = r \in \mathbb{Z}$$

We refer to r as the *ratio* of oversampling. This case is illustrated in Figure 11. In this case, for any edge $e \in g$ of the form $\tau_1 \rightarrow \tau_2$, we require a flow instance to have the form

$$\tau_1(i) \rightarrow \tau_2(\lfloor i * r + k \rfloor) \text{for } k \in \{0, \ldots, r-1\}$$

or

$$\tau_1(i) \rightarrow \tau_2(\lfloor (i+1) * r + k \rfloor) \text{for } k \in \{0, \ldots, r-1\}$$

Again, for repeated execution of a fixed frame, the index can wrap around into the next frame. This contraint will enforce the integrity of single-buffering.

Undersampling occurs when a flow contains the edge $\tau_1 \rightarrow \tau_2$ and $T(\tau_1) \leq T(\tau_2)$. In this case we require that the under sampling rate is an integer, that is

$$\frac{T(\tau_2)}{T(\tau_1)} = r \in \mathbb{Z}$$

In this case, any edge $e \in g$ of the form $\tau_1 \rightarrow \tau_2$, we require a flow instance to have the form

$$\tau_1(i) \rightarrow \tau_2(\lfloor i/r \rfloor)$$

or

$$\tau_1(i) \rightarrow \tau_2(\lfloor i/r \rfloor + 1)$$

As above, for repeated execution of a fixed frame, the index can wrap around into the next frame, and this contraint will enforce the integrity of single-buffering.

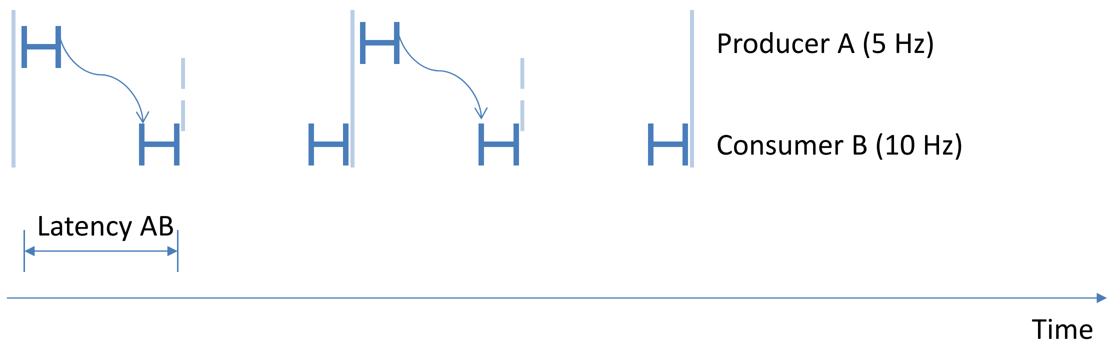

Figure 11: Oversampling when the flow period is greater than the period of task B

4.6 Schedules and Assignments

Schedules and resource allocations are our primary objectives. Schedules define the timing properties of the entire system by specifying when task instances and partition activations start and stop. In parallel to schedules, resource allocations define *where* those tasks run. In some cases these allocations hold only while the task is executing, and so the resource can be allocated to other tasks at other times. In other cases, the resource is allocated *staticly*, meaning that it is allocated for a single use, across the entire schedule. In Section 5, we define the constraints that may be applied to a given schedule and the corresponding resource allocation.[6]

Given the formal structure we have defined above, we can define two different types of schedules: task schedules and partition schedules. Task schedules are a good test case for timing properties of real time systems, while partition schedules are required for industrial problems such as ARINC 653 scheduling.

Definition 4.13 (Task Schedule) *A static task schedule is a function which maps tasks to times.* $\sigma : \{\mathcal{T} \cup \emptyset\} \to 2^{\mathbb{Z}_{\geq 0}}$

This definition maps *tasks* to subsets of the timeline, consisting of disjoint intervals, each of which corresponds to the execution of an instance of a given task.

The idea of a schedule is lacking in regards to knowing *where* (on what system) a task is running. To address this we define a task allocation.

Definition 4.14 (Task Allocation) *An* task allocation *is a mapping of tasks to systems* $\alpha : \mathcal{T}_{inst} \to \mathcal{S}$.

The particular resources on which a task is running changes the context switch time of that task. Allocating tasks to systems and scheduling them induces context switch overhead which is not allotted for in the compute time of a task or task instance. The context switch overhead of a task then depends on both on the resource allocation α of τ and on the schedule of τ (more preemptions = more overhead). In order to abstract away the details of task context switching, we define context switch time as a function $\Delta_{\sigma,\alpha}$.

Definition 4.15 *The* context switch *overhead of task instance* $\tau(j)$ *under a schedule* σ *and resource allocation* α *is a function* $\Delta_{\sigma,\alpha} : \mathcal{T}_{inst} \to \mathbb{Z}_{\geq 0}$. *We refer to the minimum context switch time as* $\Delta = \min \Delta_{\sigma,\alpha}$

[6]Hereafter, when there is no need to distinguish, we will refer to these simply as a schedule, with the allocation implicit.

In this general setting, tasks can be scheduled for arbitrary subsets of times, however it will often be useful to talk about slices scheduled for continuous segments of time. We call these segments of time *slices* and define them formally

Definition 4.16 (slice) *A* slice *of task instance $\tau(j)$ is an interval on $[a, b) \subset \mathbb{Z}$ such that $[a, b] \cap \sigma(\tau(j)) = [a, b]$. Unless otherwise noted, we assume a slice to be* maximal, *meaning that $a - 1 \notin \sigma(\tau(j)$ and $b \notin \sigma(\tau(j)$.*

Example 4.17 *A typical way to visualize a schedule is in a table such as this one*

$$\left(\begin{array}{lcccccccccccccccc}
\textit{time} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\textit{cpu0} & \tau_1(1) & - & \tau_2(1) & \tau_2(1) & \tau_2(1) & \tau_2(1) & - & - & - & \tau_1(2) & - & - & - & - & - & - \\
\textit{mem0} & \tau_1(1) & - & \tau_2(1) & \tau_2(1) & \tau_2(1) & \tau_2(1) & - & - & - & \tau_1(2) & - & - & - & - & - & - \\
\textit{eth0} & - & - & - & - & - & - & \tau_3(1) & \tau_3(1) & - & - & - & - & - & - & - & - \\
\textit{cpu1} & - & - & - & - & - & - & - & - & \tau_4(1) & \tau_4(1) & \tau_4(1) & \tau_4(1) & \tau_4(1) & - & - & - \\
\textit{mem1} & - & - & - & - & - & - & - & - & \tau_4(1) & \tau_4(1) & \tau_4(1) & \tau_4(1) & \tau_4(1) & - & - & -
\end{array}\right)$$

In this table, we can see that $\sigma(\tau_1(1) = \{0\}$ while $\sigma(\tau_4(1)) = \{8, 9, 10, 11, 12\}$, making $[8, 12]$ a slice of $\tau_4(1)$. We further can see that $\alpha(\tau_2(1)) = \{cpu0, mem0\}$ and $\alpha(\tau_3(1)) = \{eth0\}$. Implicitly, we also see that $\sigma(\emptyset) = \{13, 14, 15\}$.

This example also shows the utility of *resource timelines*, which constitute the rows in the table.

For convenience, the following functional notation may be used:

$$\begin{aligned}
start(\sigma, \tau(j)) &= \min(\sigma(\tau(j))) \\
finish(\sigma, \tau(j)) &= \max(\sigma(\tau(j))) + 1 \\
dur(\sigma, \tau(j)) &= |\sigma(\tau(j)) \cap \{0, \ldots, \Pi - 1\}|
\end{aligned}$$

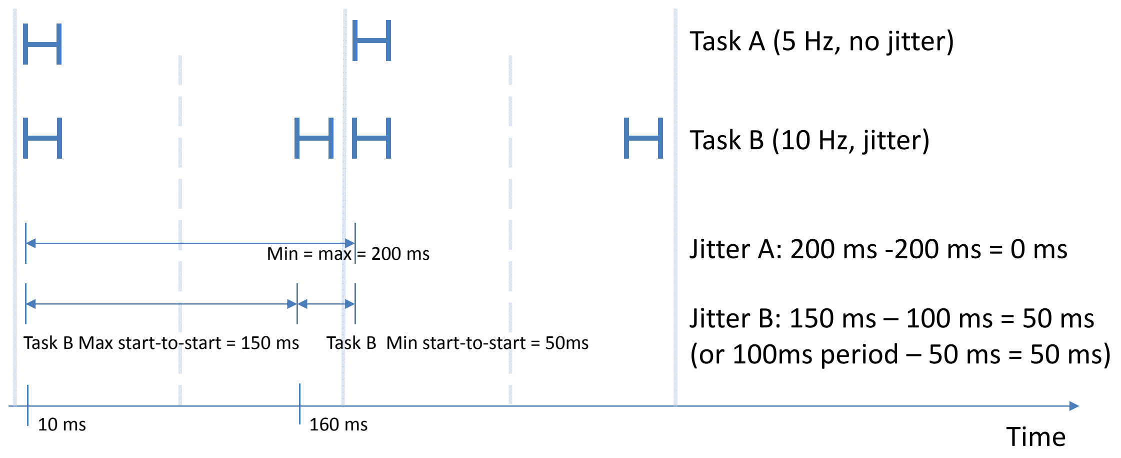
Task A (5 Hz, no jitter)
Task B (10 Hz, jitter)
Min = max = 200 ms
Jitter A: 200 ms -200 ms = 0 ms
Task B Max start-to-start = 150 ms
Task B Min start-to-start = 50ms
Jitter B: 150 ms – 100 ms = 50 ms
(or 100ms period – 50 ms = 50 ms)
10 ms
160 ms
Time

Figure 12: Example illustrating jitter.

For avionics applications using standards such as ARINC 653, it is frequently the case that tasks will be *grouped* so as to reduce resource usage, e.g., by reducing the memory footprint, or minimizing context switch time by using a light-weight scheduler among a set of related tasks. These groupings are frequently called *partitions*.

Definition 4.18 (Partition Schedule) *A* static partition schedule *σ on a set of tasks $\mathcal{T}$ split up into a set of partitions $\mathcal{P}$ having activations $\mathcal{P}_{act}$ is a function $\sigma_{\mathcal{P}} : \mathcal{P}_{act} \to 2^{\mathbb{Z}_{\geq 0}}$. As with task schedules, we implicitly extend a schedule to all of $\mathbb{Z}_{\geq 0}$ with the notion of scheduling the null task $\emptyset$.*

We will use familiar functional notation for partition schedules in which $\wp$ is a partition activation of the partition P.

$$\begin{aligned} begin(\sigma, \wp) &= \min(\sigma(\wp)) \\ end(\sigma, \wp) &= \max(\sigma(\wp)) + 1 \end{aligned}$$

Example 4.19 *We will extend Example 4.17 by assuming that tasks τ_1, τ_2,τ_4,τ_4 are partitioned as*

$$\begin{aligned} P_1 &= \{\tau_1, \tau_2\} \\ P_2 &= \{\tau_3\} \\ P_3 &= \{\tau_4\} \end{aligned}$$

These partitions are further divided up into several activations, containing task instances. The problem of dividing task instances into partition activations will be discussed in detail in Problem 6.4.

$$\begin{aligned} \wp_1(1) &= \{\tau_1(1), \tau_2(1)\} \\ \wp_1(2) &= \{\tau_1(2)\} \\ \wp_2(1) &= \{\tau_3(1)\} \\ \wp_3(1) &= \{\tau_4(1)\} \end{aligned}$$

$$\left(\begin{array}{ccccccccccccccccc}
\mathit{time} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\mathit{cpu0} & \wp_1(1) & \wp_1(1) & \wp_1(1) & \wp_1(1) & \wp_1(1) & \wp_1(1) & - & - & - & \wp_1(2) & - & - & - & - & - & - \\
\mathit{mem0} & \wp_1(1) & \wp_1(1) & \wp_1(1) & \wp_1(1) & \wp_1(1) & \wp_1(1) & - & - & - & \wp_1(2) & - & - & - & - & - & - \\
\mathit{eth0} & - & - & - & - & - & - & \wp_2(1) & \wp_2(1) & - & - & - & - & - & - & - & - \\
\mathit{cpu1} & - & - & - & - & - & - & - & - & \wp_3(1) & \wp_3(1) & \wp_3(1) & \wp_3(1) & \wp_3(1) & - & - & - \\
\mathit{mem1} & - & - & - & - & - & - & - & - & \wp_3(1) & \wp_3(1) & \wp_3(1) & \wp_3(1) & \wp_3(1) & - & - & -
\end{array}\right)$$

In this table, we can see that $\sigma(\wp_1(1)) = \{0, 1, 2, 3, 4, 5\}$ during which, τ_1 and τ_2 are allowed to run in any order (note that there is sufficient time for both to run, which we will enforce using Constraint 5.8.) In contrast $\sigma(\wp_3(1)) = \{8, 9, 10, 11, 12\}$ and can run only task $\tau_4(1)$, as it is the only task instance contained in the partition activation.

Finally, when tasks are periodic, we typically restrict our attention to *cyclic* schedules, which we have already seen examples of in Examples 4.17 and 4.19.

Definition 4.20 (Cyclic Schedule) *A schedule (partition or task schedule) is* cyclic *with* major frame Π *if*

$$\begin{aligned} t \in \sigma(\tau_i(j)) &\implies t + \Pi \in \sigma(\tau_i(j)) && \textit{for task schedules} \\ t \in \sigma(\wp) &\implies t + \Pi \in \sigma(\wp) && \textit{for partition schedules} \end{aligned}$$

In a cyclic schedule, each frame is like the any other. It suffices to describe a single frame. Modular arithmetic over frame duration will be used to express some temporal relations.

5 Constraints

In this section, we define constraints over cyclic partition schedules. In all of the following, we use this notation:

- $\mathcal{R} = \{R_i\}$ is the set of resource which contains a distinguished subset $R_{static} \subset \mathcal{R}$ of *static* resources,
- $\mathcal{S} = \{S_i\}$ is the set of systems,
- $\mathcal{T} = \{\tau_i\}$ is the set of tasks, with $\mathcal{T}_{inst}$ denoting task instances,
- $\mathcal{F} = \{\varphi_i\}$ is the set of flows, with $\mathcal{F}_{inst}$ denoting flow instances,
- $\mathcal{P} = \{\wp_i\}$ is the set of partitions, with $\mathcal{P}_{act}$ denoting partition activations,
- and (σ, α) is schedule/allocation pair.

The constraints described in this section are grouped into the following sections:

Task constraints **(Section 5.1)** - Constraints arising from the timing properties of tasks and flows, including period, jitter, duration, and latency.

Partition constraints **(Section 5.2)** - Constraints arising from the grouping of tasks, including ordering, latency, and jitter.

Resource constraints **(Section 5.3)** - Constraints arising from individual resources including over allocation, context switch, and migration which apply to both tasks and partitions.

System constraints **(Section 5.4)** - Constraints arising from systems, including checking topology, allowed systems, and migration, which apply to both tasks and partitions.

5.1 Task Constraints

Constraint 5.1 (Task Duration Constraint) *For every task instance $\tau(j)$ of $\tau \in \mathcal{T}$, and every resource $R \in \mathcal{R}$, we require:*

$$dur(\sigma, \tau(j)) \geq C(\tau) + \Delta_{\sigma,\alpha}(\tau(j)) \forall \tau(j) \in \mathcal{T}_{inst}.$$

The *Task Duration constraint* requires that all tasks are assigned adequate compute time by the schedule on their required resources. The duration may be larger than the compute time alone, as the scheduled time must also include context switch and preemption overhead.

Constraint 5.2 (Jitter Constraint) *For every task instance $\tau(j)$ of $\tau \in \mathcal{T}$,*

$$T(\tau) - J(\tau) \leq start(\sigma, \tau(j+1) - start(\sigma, \tau(j)) \leq T(\tau) + J(\tau)$$

Jitter constraints require bounds on the earliest and latest start times available to task instances.

Constraint 5.3 (Start-to-Finish Latency Constraint) *The* Start to Finish latency *ties the end points of a data flow to the start of its first task and the end of its last task. For every flow instance φ in $\mathcal{F}_{inst}$ and every $\tau_i(j) \leq \tau_k(\ell) \in \varphi$:*

$$finish(\sigma, \tau_k(\ell)) - start(\sigma, \tau_i(j)) \leq L(\varphi).$$

Start to Finish Latency Constraints require that all data flows in the schedule complete within their latency bounds. This is illustrated in Example 4.17.

Constraint 5.4 (Partial Order Flow Constraint) *For every flow instance $\varphi(j)$ of flow $\varphi \in \mathcal{F}$*

$$\begin{aligned} e \equiv \tau_i(k) \rightarrow \tau_j(l) \in E(g(\varphi)) \implies \\ start(\tau_j(k)) \geq finish(\tau_i(l)). \end{aligned}$$

Partial order flow constraint requires that task instances are only scheduled according to their order in a flow instance. In Example 4.17 this constraint prevents instances of task τ_2 from being scheduled concurrently with instances of task τ_1 or τ_3.

Constraint 5.5 (Strictly Sequential Instance Constraint) *For every task instance $\tau_i(j) \in \mathcal{T}_{inst}$*

$$finish(\tau_i(j) \leq start(\tau_i(j+1))$$

The *strictly sequential instance constraint* requires that sequential instances of a task τ do not overlap in time, i.e. instance $\tau(j+1)$ cannot start until after $\tau(j)$ has finished.

Constraint 5.6 (Minimum Task Duration) *For every task $\tau_i(j)$ (including the null task $\emptyset$) we require*

$$t \in \sigma(\tau) \implies \exists a \leq t \leq b \text{ so that } b + 1 - a > \Delta_{\sigma,\alpha}(\tau) \text{ and } \sigma(\tau_i(j)) = [a, b]$$

The *Minimum Task Duration* constraint requires that all tasks (including the null task) are not scheduled for less time than it takes to perform a context switch. This is because the processor is never truly idle and when scheduling an idle task for so short a time it would be better to switch directly to the next task. It is possible that a task finishes early (i.e. in less time than its worst case execution time). We leave the semantics of such a situation to the implementer - our objective is to provide a guarantee of sufficient time.

5.2 Partition Constraints

Constraint 5.7 (Partition Activation Constraint) *For every pair of partition activation $\wp, \wp'$ of a single partition $P \in \mathcal{P}$ and task $\tau_i \in \mathcal{T}$*

$$\text{If } \tau_i(j) \in \wp \text{ and } \tau_i(k) \in \wp' \text{ and } j \neq k \text{ then } \wp \neq \wp'$$

The partition activation constraint enforces a sanity condition on partition activations, namely that multiple instances of the same task must be separated into different partition activations. This assumes $\mathcal{P}$ to be a proper set-theoretic partitioning of tasks, i.e., each task is assigned to exactly one partition.

Constraint 5.8 (Partition Activation Duration Constraint) *For every partition activation $\wp$ of any partition $P \in \mathcal{P}$*

$$\Delta(\wp) + \sum_{\tau_i(j) \in \wp} dur(\tau_i(j))_{\wp} \leq dur(\sigma, \wp)$$

The partition activation duration constraint is analogous to the task duration constraint and is intended to provide sufficient resources for a partition activation. It guarantees that a partition activation has sufficient computational time to run all the task instances it contains, as well adequate context switch time. Notice that $dur(\sigma_R, \tau(j))$ includes the context switch from the resource R.

Constraint 5.9 (Minimum Partition Idle Constraint) *if $\tilde{\sigma}$ is a partition schedule with $\tilde{\sigma}_R(t) = \emptyset$ then for all $\wp_1, \wp_2$ with $end(\wp_1) \leq t \leq begin(\wp_2)$ we require:*

$$\Delta \leq begin(\wp_2) - end(\wp_1)$$

The minimum partition idle constraint deals with null activations, which, due to context switch, have a minimum allowed duration.

Constraint 5.10 (Begin-to-End Latency Constraint) *The* Beginning to End latency *constraint requires that for every flow instance φ in $\mathcal{F}_{inst}$ and every $\tau_i(j) \leq \tau_k(\ell) \in \varphi$:*

$$finish(\wp(\tau_k(\ell))) - begin(\wp(\tau_i(j))) - T(\tau_k) \cdot sync(\alpha(\tau_i(j)), \alpha(\tau_k(\ell))) \leq L(\varphi).$$

The *partition latency constraint* is essentially the same as the end-to-end latency constraint 5.3, with two differences. First latency is measured over partition activations, rather than task instances. Second, we have added a term to account for the possible presence of asynchronous boundaries on communication paths used by the flow instance.

Constraint 5.11 (Partition Jitter Constraint) *For every task instance $\tau(j)$ of $\tau \in \mathcal{T}$,*

$$T(\tau) - J(\tau) \leq begin(\wp(\tau(j+1))) - begin(\wp(\tau(j))) \leq T(\tau) + J(\tau)$$

The *partition jitter constraint* is similar to the task jitter constraint 5.2. However, jitter is measured relative to the start times of the partition activations containing consecutive instances of τ.

Constraint 5.12 (Valid Partitioning) *The set $\mathcal{P}$ of all partitions is a set theoretic partition of the set of all tasks. That is*

$$\emptyset \notin \mathcal{P}$$

$$\bigcup_{P \in \mathcal{P}} P = \mathcal{P}$$

$$P \cap P' = \emptyset \text{ for all } P, P' \in \mathcal{P}.$$

This constraint ensures the well-behaved nature of partitions discussed in 4.8, namely that every task is contained in exactly one partition.

Constraint 5.13 (Sequential Activation Constraint) *For any two partition activations $\wp, \wp'$ of P*

$$E(\wp) \leq B(\wp') \text{ or } E(\wp') \leq B(\wp)$$

The *sequential activation constraint* requires that no two activations of the same partition can overlap. This would be a self-preemption and a significant fault.

5.3 Resource Constraints

Constraint 5.14 (The quantity constraint) *For every resource R, at all time t, we require that*

$$\sum_{t \in \sigma(\tau_i(j))} Q_{\tau_i(j)}(R) \leq \xi(R)$$

This says that the quantity of a resource used at any time t may never exceed the total availability of that resource in the system.

Constraint 5.15 (The static quantity constraint) *For any fluent resource $R \in R_{static}$*

$$\sum_{\tau_i(j) | R \in \alpha(\tau_i(j))} Q_{\tau_i(j)}(R) \leq \xi(R)$$

This constraint allows us to model things like memory usage, which must be allocated at all times, whether or not the task is executing at those times.

Constraint 5.16 (The non-migration constraint) *For every resource $R \in \mathcal{R}$ and two instances $\tau_i(j), \tau_i(k)$ of a task τ_i, we require that if $R \in \alpha(\tau(j))$ then $R \in \alpha(\tau(k))$*

The *resource non-migration* constraint says that once a task is allocated to a resource, it is not allowed to be run on any other resource. That is, two instances of a single task cannot run on two different resources. We acknowledge that this restricts the class of allowed schedules but feel it necessary in order to provide guarantees and maintain our own sanity.

5.4 System Constraints

Constraint 5.17 (Connectivity Constraint) *For every flow instance $\varphi(k)$ of $\varphi \in \mathcal{F}$, if $\tau_i(\cdot) \rightarrow \tau_j(\cdot)$ is an edge in $\varphi(j)$, we require the existence of a system S_k so that*

$$\alpha(\tau_i(\cdot)) \subset S_k$$
$$\alpha(\tau_j(\cdot)) \subset S_k$$

The *Connectivity Constraint* requires that tasks are assigned to resources in such a manner that successive elements in the flow are on systems share a resource. In Example 4.17, this constraint asserts that, within the flow the system on which τ_1 runs must be provide an *eth* resource and the system on which τ_2 is run must be connected to a system with both *cpu* and *mem* resources.

Constraint 5.18 (Non-migration Constraint) $\alpha(\tau(i)) = \alpha(\tau(j))$ *for all i, j.*

The *Non-migration Constraint* confines all instances of a given task to run on the same system. It further gives us license to write $\alpha(\tau)$ and talk about the resource assignment of a task τ rather than the resource assignment of a task instance $\tau(j)$.

Constraint 5.19 (Allowed Resource Constraint) *For every task $\tau \in \mathcal{T}$, $\alpha(\tau) \in \mathcal{A}(\tau)$.*

This constraint requires simply that the resource assigned to τ be one of those that is allowed for τ.

6 Defining the Scheduling Problem for SPICA

In this section, we define the scheduling problem addressed in SPICA. This scheduing problem is static, cyclic, partition scheduling with topological constraints and (possibly) asynchronous boundaries. We continue to use the notation defined in Section 5, specifically:

- $\mathcal{R} = \{R_i\}$ is the set of resources, including a distinguished subset $R_{static} \subset \mathcal{R}$ of *static* resources,
- $\mathcal{S} = \{S_i\}$ is the set of systems,
- $\mathcal{T} = \{\tau_i\}$ is the set of tasks, with $\mathcal{T}_{inst}$ denoting task instances,
- $\mathcal{F} = \{\varphi_i\}$ is the set of flows, with $\mathcal{F}_{inst}$ denoting flow instances,
- $\mathcal{P} = \{\wp_i\}$ is the set of partitions, with $\mathcal{P}_{act}$ denoting partition activations,
- and (σ, α) is schedule/allocation pair.

Definition 6.1 (Resource Assignment Problem) *A schedule σ_R with resource assignment function α is said to satisfy the resource assignment problem if it satisfies all of the following constraints*

1. *The allowed systems constraint (Definition 5.19),*
2. *The Quantity constraint (Definition 5.14),*
3. *The Static Quantity constraint (Definition 5.15),*
4. *The non-migration constraint (Definition 5.18).*

Definition 6.2 (Topological Resource Assignment Problem) *A schedule σ_R with resource assignment function α is said to satisfy the topological resource assignment problem if it satisfies all of the following constraints*

1. *The allowed systems constraint (Definition 5.19),*
2. *The Quantity constraint (Definition 5.14),*
3. *The Static Quantity constraint (Definition 5.15),*
4. *The non-migration constraint (Definition 5.18),*
5. *The connectivity constraint (Definition 5.17),*

Definition 6.3 (Task Scheduling Problem) *A schedule σ_R with resource assignment function α is said to satisfy the task scheduling problem satisfies all of the following constraints*

1. *Compute time constraint (Definition 5.1),*
2. *Strictly sequential instance constraints (Definition 5.5),*
3. *Partial-order flow constraint (Definition 5.4),*
4. *Allowed Resources Constraints (Definition 5.19),*
5. *Connectivity Constraints (Definition 5.17),*
6. *Finish-to-finish latency constraints (Definition 5.10),*
7. *Jitter constraints (Definition 5.2),*

Definition 6.4 (Partition activation assignment problem)

Definition 6.5 (Partition Scheduling Problem) *A schedule σ_R with resource assignment function α is said to satisfy the partition scheduling problem if it satisfies all of the following constraints*

1. *Compute time constraint (Definition 5.1),*
2. *Strictly sequential instance constraints (Definition 5.5),*
3. *Partial-order flow constraint (Definition 5.4),*
4. *Allowed Resources Constraints (Definition 5.19),*
5. *Connectivity Constraints (Definition 5.17),*

6. End-to-end latency constraints (Definition 5.10),
7. Jitter constraints (Definition 5.11),

Notice in particular that the difference between the task scheduling problem defined in 6.3 and the partition scheduling problem defined in 6.5 is the replacement of task jitter and latency with partition jitter and latency.

7 Examples

In this section, we provide some additional examples of using the formal notation developed in previous sections to define scheduling problems. In particular, we add both flows and more complex systems.

Here is a simple example of a problem involving flows and end-to-end latency

Example 7.1

$$\mathcal{A} = \left\{ \begin{array}{r} C = (\text{“}processor\text{”}, 0, 0), \\ B = (\text{“}ethernet\text{”}, 0, 0) \end{array} \right\}$$

$$\mathcal{T} = \left\{ \begin{array}{r} \tau_1 = (5, 2, 0, (\text{“}processor\text{”}, 1)), \\ \tau_2 = (5, 1, 0, (\text{“}processor\text{”}, 1)), \\ \tau_{c_1} = (5, 1, 0, (\text{“}ethernet\text{”}, 1)), \\ \tau_{c_2} = (5, 1, 0, (\text{“}ethernet\text{”}, 1)), \end{array} \right\}$$

$$\mathcal{F} = \left\{ \begin{array}{l} \mathcal{F}_1 = ((\tau_1 \rightarrow \tau_{c_1} \rightarrow \tau_2), 4, 5), \\ \mathcal{F}_2 = ((\tau_2 \rightarrow \tau_{c_2} \rightarrow \tau_1), 4, 5) \end{array} \right\}$$

Here is a slightly more complex example, involving sub-period latency constraints. A typical RMA scheduler will not be able to schedule this.

Example 7.2

$$\mathcal{A} = \left\{ \begin{array}{r} C = (\text{“}processor\text{”}, 0, 0), \\ B = (\text{“}ethernet\text{”}, 0, 0) \end{array} \right\}$$

$$\mathcal{T} = \left\{ \begin{array}{r} \tau_1 = (10, 2, 0, (\text{“}processor\text{”}, 1)), \\ \tau_2 = (10, 2, 0, (\text{“}processor\text{”}, 1)), \\ \tau_3 = (10, 1, 0, (\text{“}processor\text{”}, 1)), \\ \tau_4 = (10, 2, 0, (\text{“}processor\text{”}, 1)), \\ \tau_5 = (10, 1, 0, (\text{“}processor\text{”}, 1)), \\ \tau_{c_1} = (10, 1, 0, (\text{“}ethernet\text{”}, 1)), \\ \tau_{c_2} = (10, 1, 0, (\text{“}ethernet\text{”}, 1)), \\ \tau_{c_3} = (5, 1, 0, (\text{“}ethernet\text{”}, 1)) \end{array} \right\}$$

$$\mathcal{F} = \left\{ \begin{array}{l} \mathcal{F}_1 = ((\tau_1 \rightarrow \tau_{c_1} \rightarrow \tau_2 \rightarrow \tau_{c_2} \rightarrow \tau_3), 8, 10), \\ \mathcal{F}_2 = ((\tau_4 \rightarrow \tau_{c_3} \rightarrow \tau_5), 9, 10) \end{array} \right\}$$

Here is an example that includes latency and jitter, and a simple definition of a system.

Example 7.3

$$\mathcal{A} = \left\{\begin{array}{r} C = (\text{“}\textit{processor}\text{”}, 0, 0), \\ B = (\text{“}\textit{ethernet}\text{”}, 0, 0) \end{array}\right\}$$

$$\mathcal{S} = \{\{C, B\}\}$$

$$\mathcal{T} = \left\{\begin{array}{r} \tau_1 = (10, 1, 0, \{\text{“}\textit{processor}\text{”}\}), \\ \tau_2 = (10, 2, 0, \{\text{“}\textit{processor}\text{”}\}), \\ \tau_3 = (10, 1, 0, \{\text{“}\textit{processor}\text{”}\}), \\ \tau_4 = (5, 1, 1, \{\text{“}\textit{processor}\text{”}\}), \\ \tau_5 = (5, 1, 1, \{\text{“}\textit{processor}\text{”}\}), \\ \tau_6 = (20, 1, 0, \{\text{“}\textit{processor}\text{”}\}), \\ \tau_{c_1} = (10, 1, 0, \{\text{“}\textit{ethernet}\text{”}\}), \\ \tau_{c_2} = (10, 1, 0, \{\text{“}\textit{ethernet}\text{”}\}), \\ \tau_{c_3} = (10, 1, 0, \{\text{“}\textit{ethernet}\text{”}\}) \end{array}\right\}$$

$$\mathcal{F} = \left\{\begin{array}{l} \mathcal{F}_1 = ((\tau_1 \to \tau_{c_1} \to \tau_2 \to \tau_{c_2} \to \tau_3), 9, 10), \\ \mathcal{F}_2 = ((\tau_4 \to \tau_{c_3} \to \tau_5), 4, 5) \end{array}\right\}$$

A sample solution for this problem instance is

$$\sigma_P = [6, 1, 4, 2, 2, 5, 3, 4, -, -, 5, 1, 4, 2, 2, 5, 3, 4, -, 5]$$
$$\sigma_B = [-, -, 1, -, 3, 2, -, -, -, 3, -, -, 1, -, 3, 2, -, -, 3, -]$$

Finally, here is a small example that captures the full complexity of the model, including context switch time, preemption time, jitter requirements, end-to-end latency. In order to schedule this, all resources must be used to near their full capacity.

Example 7.4

$$\mathcal{A} = \left\{\begin{array}{r} C_1 = (\text{“}\textit{processor}\text{”}, 10, 5), \\ C_2 = (\text{“}\textit{processor}\text{”}, 10, 5), \\ M_1 = (\text{“}\textit{memory}\text{”}, 5, 5) \\ M_2 = (\text{“}\textit{memory}\text{”}, 5, 5) \\ M_3 = (\text{“}\textit{memory}\text{”}, 5, 5) \\ M_4 = (\text{“}\textit{memory}\text{”}, 5, 5) \\ B = (\text{“}\textit{ethernet}\text{”}, 25, 25) \end{array}\right\}$$

$$\mathcal{S} = \{\{C_1, B, M_1, M_2\}\,\{C_2, B, M_3, M_4\}\}$$

$$\mathcal{T} = \left\{ \begin{array}{rl}
\tau_1 &= (1000, 100, 0, \{\textit{"processor"}, \textit{"memory"}\}), \\
\tau_2 &= (1000, 200, 0, \{\textit{"processor"}, \textit{"memory"}\}), \\
\tau_3 &= (1000, 100, 0, \{\textit{"processor"}, \textit{"memory"}\}), \\
\tau_4 &= (500, 100, 100, \{\textit{"processor"}, \textit{"memory"}\}), \\
\tau_5 &= (500, 100, 100, \{\textit{"processor"}, \textit{"memory"}\}), \\
\tau_6 &= (2000, 100, 0, \{\textit{"processor"}, \textit{"memory"}\}), \\
\tau_{c_1} &= (1000, 100, 0, \{\textit{"ethernet"}\}), \\
\tau_{c_2} &= (1000, 100, 0, \{\textit{"ethernet"}\}), \\
\tau_{c_3} &= (1000, 100, 0, \{\textit{"ethernet"}\}), \\
\hline
\tau_7 &= (1000, 100, 0, \{\textit{"processor"}, \textit{"memory"}\}), \\
\tau_8 &= (1000, 200, 0, \{\textit{"processor"}, \textit{"memory"}\}), \\
\tau_9 &= (1000, 100, 0, \{\textit{"processor"}, \textit{"memory"}\}), \\
\tau_{10} &= (500, 100, 100, \{\textit{"processor"}, \textit{"memory"}\}), \\
\tau_{11} &= (500, 100, 100, \{\textit{"processor"}, \textit{"memory"}\}), \\
\tau_{12} &= (2000, 100, 0, \{\textit{"processor"}, \textit{"memory"}\}), \\
\tau_{c_4} &= (1000, 100, 0, \{\textit{"ethernet"}\}), \\
\tau_{c_5} &= (1000, 100, 0, \{\textit{"ethernet"}\}), \\
\tau_{c_6} &= (1000, 100, 0, \{\textit{"ethernet"}\}), \\
\hline
\tau_{12} &= (2000, 100, 0, \{\textit{"processor"}, \textit{"memory"}\}), \\
\tau_{13} &= (2000, 100, 0, \{\textit{"processor"}, \textit{"memory"}\}), \\
\tau_{c_7} &= (2000, 100, 0, \{\textit{"ethernet"}\}), \\
\tau_{c_8} &= (2000, 100, 0, \{\textit{"ethernet"}\}), \\
\hline
\tau_{14} &= (2000, 100, 0, \{\textit{"processor"}, \textit{"memory"}\}), \\
\tau_{15} &= (2000, 100, 0, \{\textit{"processor"}, \textit{"memory"}\}), \\
\tau_{c_9} &= (2000, 100, 0, \{\textit{"ethernet"}\}), \\
\tau_{c_{10}} &= (2000, 100, 0, \{\textit{"ethernet"}\}),
\end{array} \right\}$$

$$\mathcal{F} = \left\{ \begin{array}{l}
\mathcal{F}_1 = ((\tau_1 \to \tau_{c_1} \to \tau_2 \to \tau_{c_2} \to \tau_3), 900, 1000), \\
\mathcal{F}_2 = ((\tau_4 \to \tau_{c_3} \to \tau_5), 400, 500) \\
\mathcal{F}_3 = ((\tau_7 \to \tau_{c_4} \to \tau_8 \to \tau_{c_5} \to \tau_9), 900, 1000), \\
\mathcal{F}_4 = ((\tau_{10} \to \tau_{c_6} \to \tau_{11}), 400, 500) \\
\mathcal{F}_5 = ((\tau_{12} \to \tau_{c_7} \to \tau_{13}), 400, 2000), \\
\mathcal{F}_6 = ((\tau_{13} \to \tau_{c_8} \to \tau_{12}), 400, 2000), \\
\mathcal{F}_7 = ((\tau_{14} \to \tau_{c_9} \to \tau_{15}), 400, 2000), \\
\mathcal{F}_8 = ((\tau_{15} \to \tau_{c_{10}} \to \tau_{14}), 400, 2000)
\end{array} \right\}$$

8 Conclusion and Next Steps

Our Phase I project addressed the primary risks of the SPICA approach, including complexity-driven scaling in domain-specific solver performance and the interaction between problem requirements (e.g., separation constraints) and the allocation flexibility provided by a given avionics architecture. In the Phase I effort we have generated a formal specification of the complete set of constraints, sufficient to represent a wide range of different avionics architectures and problems. We have produced a large set of test problems for input to the `yices` SMT solver, demonstrating the use of those constraints, along with output results and performance data, as well as a tunable test problem generator, automatically generating problem instances in `yices` input format. Finally, we have also produced an exemplar aircraft avionics architecture, rendered in both diagrams and AADL.

In Phase II, we will further extend the modeling capabilities of the SPICA system and mature the existing TRL 3 prototype to TRL 5 by improving the system's robustness, integration, and scaling performance. We have also proposed an optional task, to integrate SPICA with other AADL-based design and development tools by providing an interface (i.e., a plugin) to the Open Source AADL Tool Environment (OSATE). This will make SPICA available to the large and growing community using AADL as a modeling language for system design, as well as provide two-way access between SPICA and a wide range of open-source tools for analyzing systems specified in AADL.

Specific Phase II objectives for SPICA are divided between additional research issues to address, and maturation of the existing prototype implementation. For research, our specific goals include:

- Addressing some additional issues in modeling multi-core performance, specifically including contention for on-board cache memory under different allocation schemes.
- Integration of multiple scheduling approaches in the same model, for example supporting the use of schedulability analysis *during the scheduling process* to drive allocation decisions.
- Modeling other communication and resource allocation protocols, extending the currently-defined set of constraints as required.

For maturation of the prototype, specific objectives include:

- Improving scaling performance as discussed above.
- Finalizing the translation from AADL to SMT input format.
- Implementing a broader set of avionics architectures in our AADL model database
- Providing an explanatory capability for unschedulable models, based on the sets of constraints found to be inconsistent.

References

[1] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput. 1*, 1 (Jan. 2004), 11–33.

[2] Buttazzo, G. C. *Hard real-time computing systems.* Springer, 2005.

[3] Carpenter, T., Driscoll, K., Hoyme, K., and Carciofini, J. Arinc 659 scheduling: Problem definition. *RTSS 1994* (1994).

[4] Crawford, J. M., and Baker, A. B. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the twelfth national conference on Artificial intelligence (vol. 2)* (Menlo Park, CA, USA, 1994), AAAI'94, American Association for Artificial Intelligence, pp. 1092–1097.

[5] Feiler, P. H., Hansson, J., de Niz, D., and Wrage, L. System architecture virtual integration: An industrial case study. Tech. Rep. CMU/SEI-2009-TR-017, Software Engineering Institute, November 2008.

[6] ISO/IEC, JTC1/SC7, 42, W., and IEEE. Systems and software engineering Architecture description. http://www.iso-architecture.org/42010/, 2010. [Online; accessed 14-May-2013].

[7] Johnston, S., Edman, R., Mohammed, A., Carpenter, T., and Nelson, K. MiCART Technology Demonstration Video, 2012. http://www.adventiumlabs.com/video/micart.

[8] Liu, J. *Real-Time Systems.* Prentice Hall, 2000.

[9] Moura, L., and Bjørner, N. Satisfiability modulo theories: An appetizer. In *Formal Methods: Foundations and Applications*, M. V. Oliveira and J. Woodcock, Eds. Springer-Verlag, Berlin, Heidelberg, 2009, pp. 23–36.

[10] Steiner, W. An evaluation of smt-based schedule synthesis for time-triggered multi-hop networks. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st* (2010), IEEE, pp. 375–384.